TESTING A PROCESSOR USING A RANDOM CODE GENERATOR

Cross Reference To Related Application(s)

This application is a continuation application of copending application number 09/510,371, filed February 22, 2000, which is hereby incorporated by reference in its entirety.

Technical Field

The technical field is design verification of central processing units that execute programs according to Instruction Set Architectures.

Background

Instruction set architectures (ISA) specify how central processing units (CPU) execute programs. The programs include instructions stored in memory. Instructions are typically simple, primitive operations such as add, branch on condition, load from memory, and store to memory.

To provide for software compatibility with legacy computer systems, modern CPUs should adhere to the ISA with minimal defects. To achieve this goal, computer designers could verify the correct behavior of each kind of instruction in isolation. However, this may not be sufficient because, in an effort to improve performance, modern CPUs may overlap the execution of numerous instructions. In this environment, some defects may be exposed only when specific combinations of instructions are executed in sequence. Thus, the designer may desire to test every combination or sequence of instructions. However, modern CPUs typically execute over 100 instructions, each of which can include numerous options. Mechanically enumerating all of these possible instructions sequences may not feasible. Even if all possible instruction sequences were enumerated, the designer's test regime may be insufficient because many defects require more than just a specific sequence of instructions. In particular, additional events, such as operand data patterns, memory address collisions between instructions, the state of CPU structures such as cache memories, and stimulus received by the CPU can pre condition defects.

One of the most important tools used by a CPU designer to address these challenges is a Random Code Generator (RCG). The RCG creates a random software program that is used during the CPU design and prototype verification process to generate billions of random instructions to test the CPU.

The RCG creates the initial state of the CPU for each random software program. The initial state includes instructions in memory, and data used by these instructions. The data can be located in memory or CPU structures like registers and caches. An expected final state is also required to check the actual final state created by running the random program on the CPU. The expected final state can be created by the RCG. Alternatively, the expected final state can be obtained by running the random software program on a separate CPU reference emulator.

The RCG maintains two models of the state of the CPU registers: the initial state and the current state. The CPU registers and memory are collectively referred to as data locations. In operation, a typical RCG operation creates random values for all data locations and stores them in the initial and current states. The RCG then loops until a desired program length is achieved. A typical random software program length may be 100 instructions or more. A program counter is used to track the instructions in the random software program. If the program counter points to an uninitialized instruction memory, the RCG generates a random instruction and stores the initial and current states. Next, the instruction pointed to by the program counter is emulated, updating the current state. The program counters then increments and the next instruction is examined.

This method may work in some cases, but is often insufficient for generating high quality random software programs in many other cases. Examples of these cases include: 1) when it is desirable for addresses used by load/store instructions to contend for the same cache locations; 2) when it is desirable to avoid instruction operands that result in an excessive number of traps; 3) when it is desirable to select branch target addresses that cause different parts of the program to contend for the same cache locations; and 4) when it is desirable to avoid instruction operands that the ISA specifies are illegal or that may cause unpredictable results.

In current systems, the RCG can set aside a sub set of the registers for specific use such as memory addressing for load and store instructions. Instead of being given random values, these registers can be given desirable values. But these registers cannot be changed by the random software program because they would be overwritten with random values. This makes it impossible for the random software program to test use of these registers after an earlier modification of the registers. This situation provides unacceptable coverage holes.

In other current systems, when the RCG first selects the operands registers for an instruction, and those registers hold values that are undesirable, the RCG may repeat the

selection one or more times until more suitable registers are found. The problem is that the requirements for a register value to be desirable are often very specific. Further, no desirable registers may be available. If a desirable register is available, this approach would tend to select the desirable register a disproportionate number of times, thereby reducing the quality of the test coverage.

Summary

What is disclosed is a method for testing a processor using random code generation. The method begins by defining an initial state of the processor, the initial state including data locations in the processor. Defining the initial state includes, for each data location, creating an uncommitted value, setting an initial state pointer to the uncommitted value, and setting a current state pointer to the uncommitted value. Next a random instruction is generated for each data location, and is then executed, the executed random instructions forming a random program. Executing the random instructions produces current values for the data locations. The current values can be committed, uncommitted, or deferred. Then a length of the random program is determined, and if the random program length equals a desired length, a commit V (value) routine is executed. If the random program length is less than the desired length, the generating, executing, and determining steps are repeated until the desired program length is achieved.

Also disclosed is a method for testing a central processor unit, where the method includes the steps of defining data locations in the central processor unit, assigning random values to each of the data locations, the random values defining an initial state of the central processor unit, generating random instructions, executing one or more of the random instructions, wherein the execution provides current values to the data locations, and determining a current state of the central processor unit. Determining the current state of the central processor unit includes the steps of marking a value as uncommitted if a content is unknown and any desired content may be created by assigning a content to one or more other uncommitted values, marking the value as deferred if the content is unknown, and it is not possible to compute any desired content by assigning the content to one or more other uncommitted values, and marking the value as committed if the content is known.

Still further what is disclosed is a method for testing a processor using a random code generator, including the steps of defining data locations in the processor, assigning uncommitted values to each of the defined data locations, thereby defining an initial, fixed state of the processor, generating a random program comprising random

instructions, where the data locations represent one of input values and output values to the random instructions, executing the random program, committing values to desirable values in selected data locations, wherein committed values are produced, and assigning remaining uncommitted values to arbitrary values, thereby defining a final state of the processor.

Description of the Drawings

The detailed description will refer to the following figures, in which like numerals refer to like elements, and in which:

Figure 1 is a block diagram of a testing environment using a random code generator;

Figure 2 is a more detailed block diagram of the environment of Figure 1;

Figure 3 is an example of a random code sequence;

Figure 4 illustrates a graph data structure used to generate random programs in the presence of uncommitted values; and

Figures 5a -12c are flowcharts illustrating an embodiment of an operation of the random code generator.

Detailed Description

Central processing units (CPU) have instruction set architecture (ISA) that specify how the CPU executes programs. The programs include instructions stored in memory. Instructions are typically simple, primitive operations such as add, branch on condition, load for memory, and stored to memory.

During development of such a CPU, the designer must verify that the design complies with the ISA. This present difficulties because modern CPUs may attempt to overlap the execution of numerous instructions to improve performance. Because of this design option, some defects will only be exposed when specific combinations of instructions are executed in sequence. There are typically over 100 instructions, each of which can include numerous options. Therefore, mechanically enumerating all possible instructions sequences is not feasible. Even this would be insufficient because many defects require more than just a specific sequence of instructions. Operand data patterns, memory address collisions between instructions, the state of CPU structures such as cache memories, and stimulus received by the CPU on its external surfaces are examples of additional events that can pre condition defects.

A Random Code Generator (RCG) can be used to implement a testing program in these modern CPUs. The RCG essentially creates a random software program. RCGs are

used during the design and prototype verification process to generate billions of random instructions to test the CPU.

The RCG creates an initial state of the CPU for each random program. The initial state includes instructions in memory and data used by these instructions. The data can be located in memory or CPU locations like registers and caches. An expected final state is also required to check the actual final state created by running the random program on the CPU. This can be created by the RCG, or the final state can be obtained by running the random program on a separate CPU reference emulator.

Figure 1 is a block diagram of a test environment 10 that uses a random code generator (RCG) 20 to perform specific tests. The environment 10 includes a hardware device, or processor, 100. The environment 10 also includes a simulator 200 that simulates all or part of the processor 100. The RCG 20 is coupled to the processor 100 and the simulator 200, and may be used to perform tests on either device.

The functions of the RCG 20 may be separated into two or more components. For example, a RCG could be used to generate the random software program, and a separate emulator could be used to determine the correct final state of the data locations after execution of the random software program.

Figure 2 is a more detailed block diagram of the test environment 10 shown in Figure 1. In Figure 2, the processor 100 is shown in more detail. The simulator 200 may include modules or components that simulate one or all of the components (and their functions) of the processor 100. Accordingly, a detailed block diagram of the simulator is not needed, and one of ordinary skill in the art would understand how to construct such a simulator.

The processor 100 is shown coupled to the RCG 20. The processor 100 includes a memory subsystem 130, one or more integer value registers 110, one or more floating point registers 120, and one or more accumulators 140. Also included is a memory controller 150. The registers, memory and accumulators will be referred to hereafter as data locations. Other devices besides the integer value registers 110, the floating point registers, accumulators and memory may also be included in the processor 100 as data locations.

The RCG 20 generates an initial state of the processor 100. The initial state includes all registers, accumulators, memory and other components that are used in execution of the random software program. These registers, accumulators and memory are referred to as data locations. The RCG 20 will generate random values for selected

data locations. That is, the RCG 20 will generate random values for every data location the test sequence will actually use. The RCG 20 may generate the random values using mathematical routines that are well known in the art. As the RCG 20 generates the random instructions in the instruction sequence, the RCG 20 may emulate the random instruction. As the instruction sequence executes, the RCG 20 will update values in the data locations and will provided a current state of the processor 100. When the instruction sequence executes, the current state of the hardware device will deviate from the initial state.

Also included in the hardware device is a program counter 22. The program counter 22 is used to point to data locations, including data locations that are uninitialized. If the data location that the program counter 22 points to is uninitialized, the RCG 20 may write a value to that data location. Otherwise, the RCG 20 cannot change that data location. In this case, the RCG 20, rather than generating a new instruction, continues processing with the instruction provided at that data location. Whether the RCG 20 generates a new instruction or uses an existing instruction in a data location, the RCG 20 advances the state of the processor 100 according to the action of the instruction. The program counter 22 then points to a next data location, and the process continues.

Figure 3 shows an example of a code sequence generated by the random code generator. In Figure 3, a first instruction 30 determines a value r1 = r2 +r 3. A second instruction 40 determines a value r4 = r1 + r5.

Figure 4 shows a graph data structure 50 used to generate random programs in the presence of uncommitted values. The graph 50 is a logical representation of a data structure in the RCG 20. The graph 50 includes nodes 52 and 53 that correspond to instructions processed so far. Edges 61 - 67 of the graph 50 correspond to operand values. For example, the node 52 is an ADD instruction that receives two inputs 61 and 62 and produces a single output value 63. For data location defined by the ISA, two pointers are maintained, the first pointer points to a value that is held initially, and a second pointer points to a value that the data location holds currently. In the example illustrated in Figure 4, the pointers 71 and 73 correspond to an initial and a current state for the register R5. Pointers 75 and 77 correspond to an initial and a current state for the register R1.

Using the code sequence illustrated in Figure 3, the current state 77 of the register R1 is updated by the ADD instruction that adds values from registers R2 and R3 to

produce the output 63. The current value 73 of the register R5 is then added to the current value of the register R1 to produce the output 66.

The values shown in Figure and 4 as edges can each have one of three states. A value can be uncommitted if it can hold any number. The value is deferred if the value can no longer hold any number but its number is not yet known. Finally, the value can be committed if it holds a specific number.

In an embodiment, the RCG 20 makes available uncommitted data locations during the random program generation process. When a need exists for a data location to contain a specific (desirable) number, then the value in the data location is committed to that desirable number at that point in the execution of the random program. Uncommited values can propagate through one or more previous instructions. All data locations begin the random program in the uncommitted state. This initial state remains fixed, but a current state for each data location is updated as the random program is executed. When the RCG 20 has completed generating the random program, if any uncommitted data locations remain, the uncommitted data locations are committed to arbitrary numbers.

The process is illustrated with reference to Figures 5a - 12c. Figure 5a is a flowchart illustrating a high level algorithm executed by the RCG 20. The process starts in block 101. The RCG 20 then creates an uncommitted value for all data locations, block 105. Next, the initial state pointer and the current state pointer are set to the uncommitted value for all data locations, block 111. The RCG 20 then executes subroutine 121. Following subroutine 121, the program length is then checked by the RCG 20 to determine if the desired program length has been achieved, block 115. If the desired program length has not been achieved, the program counter 22 is incremented, block 117 and then the process returns to block 121. Once the desired program length has been achieved, the RCG 20 calls the commit (value V) subroutine 170 for each data location to commit the value pointed to by the current state pointer.

Figure 5b is a flowchart illustrating a subroutine 121 for execution when the program counter 22 points to an uninitialized instruction memory. The RCG 20 generates a random instruction and stores in the initial and current state values for that data location, block 131. The RCG 20 then inserts the instruction into the graph, block 141. For the destination data location of the instruction, the RCG 20 moves the current state pointer to point to the output value of the instruction, block 145. The subroutine then ends, block 146.

Figure 6 is a flowchart illustrating the commit (value V) subroutine 170. The RCG 20 determines a state of the value V, block 180. If V is committed, the process moves to block 182 and ends. If V is deferred, the RCG 20 determines the instruction that produced V, block 190. The RCG 20 then calls execute (I) subroutine 201. If V is neither committed nor deferred, the RCG 20 generates a random number X, block 195. The RCG 20 then calls commit (V, X) subroutine 210.

Figure 7 is a flowchart illustrating the commit (V, X) subroutine 210. The RCG 20 determines if an instruction I produced the value V, block 215. If I exists, then the RCG 20 calls subroutine reverse_propagate (I, X) 225 and then calls subroutine execute (I) 230. If no instruction I produced the value V (I = NULL), the RCG 20 sets the value of V to X, block 227. The RCG 20 then calls subroutine demote (V, committed) 240.

Figure 8 is a flowchart illustrating the subroutine demote (value V, state S) 240. In block 245, the RCG 20 checks if the state of V is committed. If the state of V is uncommitted or deferred, the RCG 20 sets the state of V to S, block 247. Then, for each instruction I that consumes V, the RCG 20 calls subroutine demote (I) 250. If the state of V was committed, the subroutine exits, block 246.

Figure 9 is a flowchart illustrating the execute (instruction I) subroutine 201. The RCG 20 calls the commit (V) subroutine 170 for each input value of I, block 260. Next, the RCG 20 performs an ISA operation for I, block 265. For each output value V of I, the RCG 20 sets the value of V to the result of the ISA operation of I, block 267. The RCG 20 then calls the demote (V, committed) subroutine 240.

Figure 10 is a flowchart illustrating an insert (instruction I) subroutine 280. For instruction I, the RCG 20 inserts the instruction into the graph 50, block 282. The RCG 20 uses the current state pointers for data locations consumed by I to determine which values I is attached to. The RCG 20 then creates a value V for the result of the instruction I, block 284. The initial state of the value V is uncommitted. The RCG 20 then executes the subroutine demote (I), block 300.

Figures 11a and 11b are flowcharts illustrating the subroutine reverse_propagate (instruction I, integer X) 225. The reverse_propagate subroutine is called for instruction types that can propagate uncommitted values. The reverse_propagate subroutine commits one or more input values of instruction I to values that will result in X being produced for the output of the instruction I when the subroutine execute (I) is called. Figure 11a is a flowchart illustrating the subroutine reverse_propagate (I, X) 225 when the instruction I is

a COPY instruction. The RCG 20 assigns V as an input value of I, block 314. The RCG 20 then calls the subroutine commit (V, X) 210, described above.

Figure 11b is a flowchart illustrating the subroutine reverse_propagate (I, X) 225 when the instruction I is an ADD instruction. The RCG 20 assigns V1 and V2 as input values for the instruction I, block 316. In block 320, the RCG 20 determines if V1 is uncommitted. If V1 is uncommitted, the RCG 20 calls subroutine commit (V2) 170, block 330. The RCG 20 then calls the subroutine commit (V1, X - V2._value) 210, block 340. If the value of Vi is not uncommitted, then the RCG calls the subroutine commit (V1) 170, block 350 and the subroutine commit (V2, X - V1._value), block 360.

Figures 12a - 12c are flowcharts illustrating the subroutine demote (instruction I) 250. The subroutine demote (instruction I) 250 is intended to demote the state of the output values of the instruction I depending on the capability of the instruction I to propagate uncommitted values. The definition of the subroutine demote (instruction I) 250 is differs with each instruction type. Figure 12a is a flowchart illustrating the subroutine demote (instruction I) 250 for instructions types that are not capable of propagating uncommitted values. Such instructions include DIVIDE, floating point instructions and Boolean instructions, for example. The RCG 20 assigns V to the output value of the instruction I, block 370. The RCG 20 then calls the subroutine demote (V, deferred) 240, block 372.

Figures 12b and 12c are flowcharts illustrating illustrating the subroutine demote (instruction I) 250 for COPY and ADD instructions.

Using the above-described processes, the RCG 20 can choose a data location with an uncommitted state as an operand of an instruction. The RCG 20 can then provide any desired content or value for entry into that data location. The existence of a deferred state helps preserve as many uncommitted values as possible. For example, if an uncommitted data location is selected for the operand of a complex instruction for which the RCG 20 cannot propagate the uncommitted value, the output value of that instruction must be marked as deferred rather than as uncommitted. However, the input value of that instruction remains uncommitted so that the input value can fulfill a need later in the random program generation process.

When using the RCG 20 to generate a random program, many instructions that may be included in the instruction sequence may have an immediate operand as part of the instruction. This situation may be viewed as an additional operand of the instruction

that can remain uncommitted for a time after the instruction is generated by the RCG 20. When the value becomes committed, the an opcode of the instruction may be modified to integrate the immediate value.

The RCG 20 may also accommodate conditional instructions (i.e., a Boolean condition), such as branching. For example, a conditional instruction: "If r1>r2 then branch," indicates a possible branch. The RCG 20 must know whether the condition (r1>r2) is true when the RCG 20 generates such an instruction so that the RCG 20 can predict the program flow. If r1 or r2 has an uncommitted state, a problem ensues. Committing data locations at this point may be undesirable since the data locations may have good uses in later phases of the random program.

To overcome this problem, the RCG 20 may decide whether the condition is true. The RCG 20 records a decision on the graph 50 corresponding to the branch. Later, when execute() is called for the instruction, the actual condition is checked against the desired outcome. If the actual condition and the desired condition differ, then the instruction opcode may be modified by the execute () subroutine to "flip" the sense of the condition. In the example provided above, if flipping is necessary, the instruction would become: If r1<=r2 then branch.

As the random program becomes longer, the number of processor registers in the uncommitted state decreases. This may eventually limit the length of the random program. In this case, the RCG 20 may set up a large table of uncommitted memory locations, such as in the memory 130. Periodically, the RCG 20 can emit a load instruction to transfer one of the uncommitted values into a randomly selected processor register.

The terms and descriptions used herein are set forth by way of illustration only and are not meant as limitations. Those skilled in the art will recognize that many variations are possible within the spirit and scope of the invention as defined in the following claims, and there equivalents, in which all terms are to be understood in their broadest possible sense unless otherwise indicated.